Техническое решение

по взаимодействию между <Шина> и <Клиент>

в части интеграции <Клиент\_1> и <Клиент\_2>

На 8 листах.

# Общая информация

Использование промежуточного программного обеспечения, ориентированного на сообщения (сервер ActiveMQ — далее <Шина>) - это популярный способ поддержки взаимодействия между приложениями. Основное преимущество сервера очереди сообщений (message queue - MQ) заключается в том, что он предлагает асинхронный и слабо связный путь для интеграции приложений.

В Java стандартом для подключения к серверу MQ с целью отправки или получения сообщений является JMS. Сервер ActiveMQ поддерживает список очередей, к которым приложения могут подключаться и отправлять/получать сообщения.

Назначением <Шина> является оптимизация электронного взаимодействия в рамках <Интеграции>;

Взаимодействие между <Клиент\_1> и <Клиент\_2> должно осуществляться через <Шина> и представлять собой обмен сообщениями двух видов:

- Для направления «из <Клиент\_1> в <Клиент\_2>» это сообщение вида <Заявка>

- Для направления «из <Клиент\_2> в <Клиент\_1>» это сообщения вида <Статус>.

Схема взаимодействия предусматривает реализацию на стороне <Шина> основных интеграционных функций:

1. функцию транспорта сообщений с гарантированной доставкой
2. функцию маршрутизации сообщений

Эти функции <Шина> реализованы при помощи продукта Apache ActiveMQ, для чего на стороне <Шина> созданы входящая и исходящая очереди сообщений, используемые <Клиент\_1> и <Клиент\_2> как внешний буфер сообщений.

Настоящий документ описывает механизм взаимодействия между <Клиент> и <Шина>.

# Сценарии взаимодействия <Шина> и <Клиент>

Взаимодействие между <Шина> и <Клиент> представляет собой обмен сообщениями двух видов.

Для направления «из <Клиент> в <ШИНА>» это сообщение вида «Статус»

Для направления «из <ШИНА> в <Клиент>» это сообщения вида «Заявка».

# Описание технической реализации взаимодействия <Клиент> и <Шина>)

Работа с менеджером очередей (создание соединения) производится с использованием интерфейса javax.jms.ConnectionFactory, предоставленный Jаvа-библиотекой ActiveMQ (класс ActiveMQConnectionFactory).

В первую очередь его необходимо объявить и сконфигурировать в файле конфигурации Spring

<?xml version="1.0" encoding="UTF-8"?>

<beans xmlns="http://www.springframework.org/schema/beans"

xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"

xmlns:context="http://www.springframework.org/schema/context"

xmlns:p="http://www.springframework.org/schema/p"

xsi:schemaLocation="http://www.springframework.org/schema/beans

http://www.springframework.org/schema/beans/spring-beans.xsd

http://www.springframework.org/schema/context

http://www.springframework.org/schema/context/spring-context.xsd">

<bean id="connectionFactory"

class="org.apache.activemq.ActiveMQConnectionFactory"

p:brokerURL="tcp://localhost:61616"

p:userName="smx"

p:password="smx"

/>

……….

</beans>

# Взаимодействие <Клиент> с <Шина> (передача сообщения)

В этом разделе мы покажем, каким образом отправлять сообщения с применением JMS в Spring. Для этой цели мы будем использовать класс org.springframework.jms.core.JmsTemplate.

В конфигурации Spring помимо connectionFactory необходимо объявить экземпляр JmsTemplate с аргументом конструктора connectionFactory и свойством defaultDestinationName, установленным в нужную нам очередь (в примере ”testspring”).

<bean id="jmsTemplate" class="org.springframework.jms.core.JmsTemplate">

<constructor-arg name="connectionFactory" ref="connectionFactory"/>

<property name="defaultDestinationName" value="testspring"/>

</bean>

Далее в методе sendMessage() мы вызываем метод JmsTemplate.send () с конструированием на месте экземпляра интерфейса org.springframework.jms.core.MessageCreator. В экземпляре MessageCreator реализован метод createMessage(), создающий новый экземпляр TextMessage, который будет отправлен ActiveMQ.

public void sendMessage(final String message) {

this.jmsTemplate.send(new MessageCreator() {

public Message createMessage(Session session)

throws JMSException {

return session.createTextMessage(message);

}

});

}

# Взаимодействие <Клиент> с <ШИНА> (получение сообщения)

Для разработки прослушивателя сообщений понадобится создать класс, который реализует интерфейс javax.jms.MessageListener и его метод onMessage().

public void onMessage(Message message) {

TextMessage textMessage = (TextMessage) message;

try {

logger.info("Message received: " + textMessage.getText());

} catch (JMSException ex) {

logger.error("JMS error", ex);

}

}

При поступлении сообщения методу onMessage() передается экземпляр интерфейса javax.jms.Message. Внутри этого метода сообщение приводится к экземпляру интерфейса javax.jms.TextMessage, а затем с помощью метода TextMessage. GetText() извлекается тело сообщения. Список возможных форматов сообщения приведен в онлайновой документации по JEE.

После построения прослушивателя сообщений следующий шаг заключается в определении конфигурации ApplicationContext. В листинге представлено часть содержимого файла jms-listener-app-context. xml.

<bean id="simpleMessageListener" class="ru.smartsoft.SimpleMessageListener"/>

<jms:listener-container container-type="default"

connection-factory="connectionFactory" acknowledge="auto">

<jms:listener destination="testspring" ref="simpleMessageListener" method="onMessage" />

</jms:listener-container>

Здесь мы объявляем бин типа SimpleMessageListener и используем дескриптор <jms: listener-container>, предлагаемый пространством имен jms в Spriпg, для объявления прослушивателя сообщений с указанием получателя (т.е . очереди testspring), ссылки на бин и метода, вызываемого при поступлении сообщения.

# Рекомендации по реализации транспортной подсистемы <Клиент>

Соединение с менеджером очередей осуществляется с использованием протокола STOMP http://stomp.fusesource.org/documentation/php/book.html. Для соединения вызывается функция, которая возвращает объект соединения. Во время использования STOMP не требуется отдельного открытия и поддержания ссылок на открытые очереди, метод send получает имя очереди и тело сообщения для отправки.

Необходимо посылать сообщения в режиме с получением ответа от адаптера STOMP, иначе без получения подтверждения нет возможности понять, дошло ли сообщение во входящую очередь. 4-й параметр вызова функции send - обязательно должен быть выставлен в true.

Транспортный блок необходимо реализовать в виде отдельного модуля, который инкапсулирует все механизмы связи с очередью, чтобы обеспечить: единый механизм обработки ошибок, возможность менять константы и логику соединения в одном месте для всех взаимодействий <ШИНА> -> <Клиент>.

# Форматы данных

# Номер заявки и единый номер обращения

Запрос от <Клиент\_1> к <Клиент\_2> идентифицируется уникальным номером. При поступлении сообщения в очередь исходящих сообщений для <Клиент> в <ШИНА> – **CLIENT\_1.STATUS** (сообщения обновления статуса), <ШИНА> кроме номера заявки возвращает единый номер обращения, сформированный в следующем формате:

xxxx-yyyy-zzzzzzz-nnnnnnn/yy

xxxx - код ИС <Клиент\_1> как организации, где создано обращение

yyyy– код ИС <Клиент\_2> как получателя сообщения

zzzzzz – код процесса по реестру процессов шины

nnnnnnn – порядковый номер сообщения

yy - текущий год

Примеры:

0001-0002-000101-2242460/16

<Клиент>, при получении сообщения со статусом заявки, сохраняет в БД <Клиент> присвоенный заявке единый номер обращения.

Описание реализации сервиса по назначению единого номера обращения выполнено отдельным документом «Назначение единого номера обращения (ЕНО). Техническое решение».

# Формат передачи сообщений

Сообщения, которыми обмениваются <Клиент> и <ШИНА> должны быть сформированы в формате XML.

xsd-схема формата версии V0 приведена в Приложении 1 к настоящему документу.

# Доступ к стендам

# Тестовые стенды

Тестовый стенд <ШИНА>:

IP-адрес тестового стенда: etp.sm-soft.ru

Порт тестового стенда:61613

Авторизация отключена

Тестовый стенд принимает соединения по протоколу STOMP

Тестовый стенд <Клиент>:

??? внести параметры тестового стенда <Клиент>

# Приложение 1. xsd-схема описания формата

<?xml version="1.0" encoding="UTF-8"?>